QUESTION 1

(a) What is the two-phase locking protocol? How does it guarantee serializability? (5)

The two-phase locking protocol is a protocol that enforces all the locking operations in a transaction to occur before the first unlock operation of the transaction occurs. A transaction T that satisfies the two-phase locking protocol can be divided into two phases: expanding or growing (first) phase and shrinking (second) phase. During the expanding or growing (first) phase, any locking operations on data items can be issued, but none of which can be released. On the other hand, during shrinking (second) phase, any existing locks that had been issued in the first phase can be released, but the issuance of any new locks is not valid.

Two-phase locking protocol through its implementation guarantees the serializability of the schedule. This is done by enforcing any transactions *T* in a schedule S to issue all the lock operations required for the read or writes operations of the transaction, before the issuance of first unlock operation. Since all the read_lock and write_lock on an item is issued by the transaction *T* before it needs to read or write on that item and any first unlock operation is issued, any other transaction *T'* is enforced to wait if it were to issue any conflicting lock on the same item. By that enforcement, any conflicting operation on that item is obviously impossible to happen in the interleaving operations of both *T* and *T'*. Moreover, after the first unlock operation of *T*, the read and write operation of *T* is limited to only the item that *T* currently has the lock on, since *T* are not able to issue any other lock. On the other hand, the lock operation of transaction *T'* is limited to only the item that T does not have any conflicting lock on. In other word, they can only lock and access different item, or perform non-conflicting operation on the same item. Hence, they are both conflict equivalence to any serial schedule.

(b) Describe the wait-die and wound-wait protocols for deadlock prevention. (5)

Wait-die and wound-wait are the two schemes used in deadlock prevention. Suppose that there is a transaction T_i trying to lock an item X, but the item X has been locked by the other transaction T_j using a conflicting lock. Wait-die and wound-wait protocols handle this situation in different way. Applying the ***wait-die*** protocol allows T_i to wait for T_j if T_i is older than T_j or the timestamp of the transaction T_i is smaller than the timestamp of the transaction T_j; or otherwise, aborts T_i. However, note that Ti may be restarted later in the schedule and the same timestamp will be assigned to it. Therefore, in the wait-die, only the older transaction is allowed to wait on the younger transaction.

On the other hand, applying the ***wound-wait*** protocol aborts T_j if T_i is older than T_j; or otherwise, allows T_i to wait on $T_{j.}$ In former case, T_j may be restarted later in the schedule and the same timestamp will be assigned to it. Therefore, using the wound-wait protocol, the younger transaction is allowed to wait, whereas the older transaction trying to access an item that has been exclusively accessed by the younger transaction is able to seize the access to the item by aborting the younger transaction.

(c) Describe the cautious waiting, no waiting, and timeout protocols for deadlock prevention. (5)

In addition to the wait-die and wound-wait protocol mentioned above, no waiting and cautious waiting are the other two protocols used to prevent deadlock. However, these two protocols in their implementations do not rely on the timestamp.

In the ***no waiting*** protocol, if a transaction is unable to obtain a lock, it is directly aborted. Then, after a certain time delay, the transaction will be restarted. This restart operation is however performed without any checking on whether a deadlock will actually occur or not. Hence, this protocol potentially causes the transaction to abort and restart needlessly.

Due to the drawback of the preceding protocol, cautious waiting was proposed to reduce the number of needless aborts/restarts. Suppose that the transaction Ti attempts to obtain an exclusive access (lock) on an item X, but the item X has been locked by the transaction T_j using the conflicting lock. Applying the ***cautious waiting*** protocol, the status of T_j is first checked. If T_j is blocked or waiting for some other locked item, T_i is then aborted; otherwise, T_i is blocked and allowed to wait on T_j.

The other method to deal with deadlock is to use timeouts. Using this method, if a transaction has waited for a period longer than a system-defined timeout period, the system assumes that deadlock has occurred and thus the system aborts the transaction. This action of abortion is performed regardless of whether a deadlock actually occurs or not.

(d) What is a timestamp? How does the system generate timestamp? (5)

By definition, ***timestamp*** is a unique identifier generated by the Database Management System to identify a transaction. Each time a new transaction is submitted to the DBMS, the system assigns a timestamp value to the transaction as to identify the transaction start time. Therefore, by assigning timestamp to each transaction, the operating transactions are ordered in the time order at which each transaction has been submitted to the system.

There are several methods that can be used to generate timestamps. One method is to use a counter. Using this method, the transaction timestamps are number 1, 2, 3, and so on, and thus the counter has to be incremented each time its value is assigned to a transaction. The maximum value of a computer counter however is finite, and therefore the system must reset the counter to zero whenever no transactions are submitted for some short period of time. The other method to generate timestamp is to use the current date/time value of the system clock, and ensure that no two timestamp values are generated in the same tick of the clock.

QUESTION 2

Modify the data structure for multiple-mode locks (shared and exclusive) and the algorithm for read_lock(X), write_lock(X), and unlock(X) so that upgrading and downgrading of locks are possible. (The lock and unlock operations can be found on the course page under "figures for Chapter 20"; Hint: The lock needs to check the transaction id(s) that hold the lock, if any.) (20)

In order to allow the lock conversion, the record structure of each lock needs to include the transaction identifier as to identify the locking transaction.

The transaction identifier is required to identify the transaction that requests for upgrading or downgrading. In the case of lock downgrading, the system checks if the transaction requesting for lock downgrading on an item is the transaction that is currently holding write_lock on that item. Hence, in the case of write-locked item, the request for read_lock can only be granted if it is issued by the transaction currently holding the write_lock. On the other hand, in the case of lock upgrading, the system checks if the transaction requesting for lock upgrading on an item is the only transaction that is currently holding the read_lock on that item.
As the system needs to identify the requesting transaction for each request, one extra parameter is needed on the following three algorithms: `id(T)`. The value of `id(T)` receives the value of the requesting transaction identifier. Hence, checking on the transaction identifier is performed before the system grants the request of read_lock or write_lock. Furthermore, in order to maintain the consistency of the lock records, the system adds the transaction identifier to the locking transaction list, at each time it grants the request of read_lock or write_lock to a transaction; whereas, the system removes the transaction identifier from the locking transaction list, at each time the transaction releases a lock.

The followings are the modified algorithms for read_lock, write_lock, and unlock.

Read_lock algorithm

```
read_lock(X, id(T)):
B: if LOCK(X) = "unlocked"
      then begin LOCK(X) ← "read-locked"
            no_of_reads(X) ← 1
            add id(T) to locking_transaction(X)
      end
   else if LOCK(X) = "read-locked"
      then
            no_of_reads(X) ← no_of_reads(X) + 1
            add id(T) to locking_transaction(X)
      end
   else if LOCK(X) = "write-locked"
      then  begin check id(T)
            if (id(T) = locking_transaction(X))
                  then begin LOCK(X) ← "read-locked"
                        no_of_reads(X) ← 1
                  else begin wait(until LOCK(X) = "unlocked" and
                  the lock manager wakes up the transaction);
                  go to B
            end
      end;
```

Write_lock algorithm

```
write_lock(X, id(T)):
B: if LOCK(X) = "unlocked"
      then begin LOCK(X) ← "write-locked"
            add id(T) to locking_transaction(X)
   else begin
      if(no_of_reads = 1 and id(T) = locking_transaction(X))
            then begin LOCK(X) ← "write-locked";
            no_of_reads(X) ← 0
      else begin wait(until LOCK(X) = "unlocked" and the lock
      manager wakes up the transaction);
      go to B
      end;
end;
```

Unlock algorithm

```
unlock(X, id(T))
if LOCK(X) = "write-locked"
      then begin LOCK(X) ← "unlocked"
            remove id(T) from locking_transaction(X)
            wakeup one of the waiting transactions, if any
      end
else if LOCK(X) = "read-locked"
      then begin
            no_of_reads(X) ← no_of_reads(X) - 1;
            remove id(T) from locking_transaction(X)
            if no_of_reads(X) = 0
                  then begin LOCK(X) = "unlocked";
                        wakeup one of the waiting transactions, if
                        any
                        end
            end;
```

QUESTION 3

Apply the timestamp ordering algorithm to the schedules of Figure 19.8(b) and (c), and determine whether the algorithm will allow the execution of the schedules. (the figures can be found on the course page under "figures for chapter 19".) (20)

Timestamp Ordering (TO) Algorithm - in detail

If T issues **write_item**(X) **then**

if (read_TS(X) > TS(T) or write_TS(X) > TS(T)) **then** abort T

otherwise (*TS(T) ≥ read_TS(X) and TS(T) ≥ write_TS(X)*)

```
        execute write_item(X)
        set write_TS(X) to TS(T)

  **if** T issues **read_item**(X) **then**

    **if** write_TS(X) > TS(T) **then** abort T

**otherwise** (*TS(T) ≥ write_TS(X)*)
        execute read_item(X)
        set read_TS(X) to max{TS(T), read_TS(X)}
```

Assume that initially the time stamps for all the data items are set to be 0. In addition, we have TS(T2) < TS(T3) < TS(T1). (For simplicity, assume that TS(T1) = 6, TS(T2) = 1, TS(T3) = 4.)

	T1	T2	T3		
1		Read(z)		TS(T2) > W-TS(z)	R-TS(z) ← 1
2		Read(y)		TS(T2) > W-TS(y)	R-TS(y) ← 1

3		Write(y)		TS(T2) > W-TS(y)
				TS(T2) = R-TS(y)
					W-TS(y) ← 1

4			Read(y)	TS(T3) > W-TS(y)	R-TS(y) ← 4
5			Read(z)	TS(T3) > W-TS(z)	R-TS(z) ← 4
6	Read(x)			TS(T1) > W-TS(x)	R-TS(x) ← 6

7	Write(x)			TS(T1) > W-TS(x)
				TS(T1) = R-TS(x)
					W-TS(x) ← 6

8			Write(y)	TS(T3) > W-TS(y)
				TS(T3) = R-TS(y)
					W-TS(y) ← 4

9			Write(z)	TS(T3) > W-TS(z)
				TS(T3) = R-TS(z)
					W-TS(z) ← 4

10		Read(x)		TS(T2) < W-TS(x) = 1 = 6	T2 aborts.
11	Read(y)				
12	Write(y)				
13		Write(x)			

Since T2 aborts at time point 10 and T3 reads from T2 (i.e., T3 reads y at time point 4, which was written at time point 3 by T2), T3 also aborts (cascading roll back). For the same reason, T1 has to be rolled back (see time point 11.)

Assume that initially the time stamps for all the data items are set to be 0.

TS(T1) = 3, TS(T2) = 7, TS(T3) = 1.

	T1	T2	T3		
1			Read(y)	TS(T3) > W-TS(y)	R-TS(y) ← 1
2			Read(z)	TS(T3) > W-TS(z)	R-TS(z) ← 1
3	Read(x)			TS(T1) > W-TS(x)	R-TS(y) ← 3

4	Write(x)			TS(T1) > W-TS(x)
				TS(T1) = R-TS(x)
					W-TS(x) ← 3

5			Write(y)	TS(T3) > W-TS(y)
				TS(T3) = R-TS(y)
					W-TS(y) ← 1

6			Write(z)	TS(T3) > W-TS(z)
				TS(T3) = R-TS(z)
					W-TS(z) ← 1

7		Read(z)		TS(T2) > W-TS(z)	R-TS(z) ← 7
8	Read(y)			TS(T1) > W-TS(y)	R-TS(y) ← 3

9	Write(y)			TS(T1) > W-TS(y)
				TS(T3) = R-TS(y)
					W-TS(y) ← 3

10		Read(y)		TS(T2) > W-TS(y)	R-TS(y) ← 7

11		Write(y)		TS(T2) > W-TS(y)
				TS(T2) = R-TS(y)
					W-TS(y) ← 7

12		Read(x)		TS(T2) > W-TS(x)	R-TS(x) ← 7

13		Write(x)		TS(T2) > W-TS(x)
				TS(T2) = R-TS(z)
					W-TS(x) ← 7

From the above trace, we can see that all the transaction can be successfully executed.

QUESTION 4

By definition, intention-shared (IS) lock is a type of intention lock used to indicate that a shared lock(s) will be requested on some descendant node(s); whereas, intention-exclusive (IX) lock is another type of intention lock used to indicate that an exclusive lock(s) will be requested on some descendant node(s). According to the multiple granularity locking (MGL) protocol, either types of intention lock, intention-shared (IS) or intention-exclusive (IX), should only be issued on the parent whose node will be locked in the corresponding mode. The issuance of either types of lock on a specific parent node is not an explicit lock on the node itself. It is only an indicator that there will be a corresponding type of lock requested on some descendant node(s). Therefore, since the intention-shared (IS) and intention-exclusive (IX) locks only act as indicators and the issuance of either both does not explicitly lock on any item, IS and IX are compatible.

QUESTION 5

In multiversion two-phase locking protocol, certify lock is an exclusive lock issued to confirm that the modification of the currently write-locked item is to be reflected permanently. A certify lock has to be issued on all items that the transaction has hold write lock on, prior to the transaction's commit operation.

The idea behind multiversion two-phase locking protocol is to allow two transactions to hold conflicting locks – read lock and write lock – on the same item, at the same time. The realization of this idea can only be accomplished by allowing two versions for each item X. One version, X, is used to represent the original version of item that has been written by some committed transaction; whereas, the other version, X', is used to represent the copied version of the item that is currently write-locked by a transaction and written by the write operation of the transaction. Therefore, in the multiversion 2PL, one transaction, T, can write the value of X', without any conflict to the value of X that is currently read by some transaction T'.

However, once transaction T is ready to commit, certify lock must come into play. In this protocol, the certify lock act as a confirmation that any changes made on the copied version of item, X', needs to be reflected on the original version of item, X. Hence, once certify locks are acquired on an item, the original version X of the data item is set to the value of version X'. If a certify lock cannot be granted, it shows that some other transactions must be holding a read lock on at least one data item which has been changed by the current transaction. Therefore, the current transaction cannot be committed. It has to wait until all those transactions which are holding the read locks on the relevant data items have been committed.